Deriving the spirit of the law

Thesis for 7.5 ECTS by Toon Alfrink

Bachelor Kunstmatige Intelligentie, Utrecht University, 2019-07-04

Mentor: Jan Broersen

Second assessor:

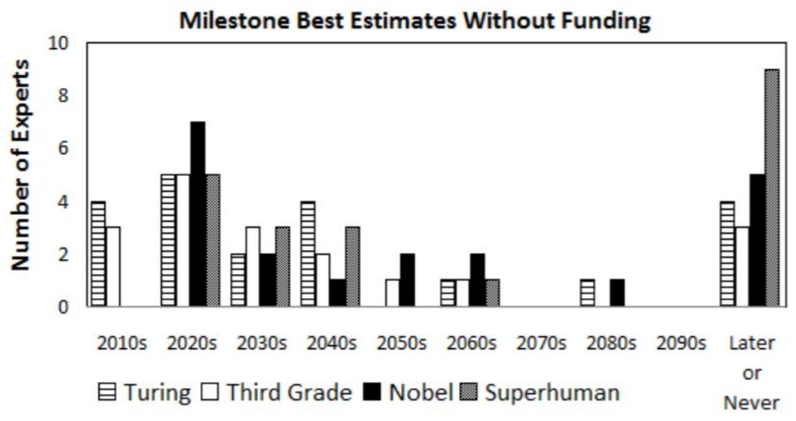
I define two approaches to rule-based AI Safety: the letter-based approach, which is to simply constrain an agent's behavior to satisfy a set of static conditions, and the spirit-based approach, which is to somehow let the agent act in accordance with what those rules intended. I explore the conditions under which a letter-based approach is insufficient. Then I describe one prominent letter-based approach to AI Safety, describe how it represents rules in STIT logic, and offer a mechanism for inferring a generalization from those rules that aims to approximate their intention. For that I use a version space learning algorithm. I finish with a small experiment.

1 Introduction

In 1956, there was a workshop in Dartmouth college. It was the first to ever be held on the topic of Artificial Intelligence. It is now considered by many to be a pivotal workshop, essentially kickstarting the field of AI. Attendants were optimistic about the possibility of creating thinking machines. Herbert Simon, one of the attendants of the workshop, was quoted saying "machines will be capable, within twenty years, of doing any work a man can do".

As that quote shows, the original intention of the field of AI was quite ambitious. Machines were expected to be able to do anything a human being could do, and perhaps even surpass them in general ability. The field had some initial successes, gaining traction by creating machines that could play checkers, do algebra and prove theorems in logic. What the researchers back in the day did not realize was that there was a substantial part of intelligence that isn't so easy to automate.

And so progress slowed down, and in 1974 the US and UK governments cut off funding for foundational AI research, ushering in a period of slow progress that would later be called the first AI winter. The field has seen ups and downs since then. Nowadays, with staggering advances[1] in the subfield of machine learning, the old promise of machines "doing any work a man can do" is again being revisited. AI systems are now beating humans in games such as poker, go, and even DotA (which is a team-vs-team roleplaying fighting game). While opinions vary about the exact date that human-level intelligence will be achieved, a survey found that the median expert expects it to happen in the 2060s.



As one can see from this figure, whether superhuman intelligence will even happen at all is debated. There is a substantial minority of experts that claim that it will never happen. For the rest of this work, we will assume that it will. This work is best read with that prospect in mind.

[1] Deepmind's AlphaZero, for instance, was able to achieve superhuman performance in Go, chess and Shogi within 24 hours of being instantiated. It learned purely by playing against itself.

1.1 The need for AI Safety

As AI systems become increasingly capable, they become increasingly influential. Already today, we allow AI to control dangerous tools like cars, medical equipment and financial assets. Not only do we give them more responsibility, they're also able to come up with strategies that are increasingly impactful. For better or for worse.

This observation has led to research efforts to make AI safe. Highlighting three approaches:

- the REINS project, which aims "to develop a formal framework for automating responsibility, liability and risk-checking for automated systems" (Broersen, 2014).
- the approach proposed by the Machine Intelligence Research Institute (MIRI), that dedicates itself to "the challenge of finding an agent architecture that will reliably and autonomously pursue a set of objectives - that is, an architecture that can at least be aligned with *some* end goal" (Soares & Fallenstein, 2014).
- The approach of openAI and the Google Brain team. Presumably partly in response to MIRI, they take a more concrete approach, discussing "the problem of accidents in machine learning systems, defined as unintended and harmful behavior that may emerge from poor design of real-world AI systems" (Amodei et al., 2016).

AI Safety is a nascent field, and researchers are not always aware of each other yet. This work is inspired by different research agendas, hoping to bring them closer together.

1.2 Two rule-based approaches to AI Safety

Let us contrast two broad approaches for securing an AI. The first is what I will call the "letter" approach. In this approach, a lawmaker secures an AI by flagging specific behaviors and outcomes as "forbidden" or "obligated". The AI follows the letter of the law. The second is the "spirit" approach, where an AI infers the rules that represent what the lawmaker actually intended, a.k.a. the spirit of the law, and follows those instead.

A lawmaker would wish that following the law to the letter leads to the same outcomes as following the law according to his intention. He may try to define the law as precisely as possible, to make sure that the letter and the spirit diverge as little as possible, or ideally don't differ at all. If they do differ in some case, we call that a "loophole". Better laws have less loopholes, and a law without any loopholes is perfect.

In practice, it seems nearly impossible to define a perfect law. Humans are not flawless at capturing exactly what they want. This reality is why I argue that the letter approach to AI Safety is incomplete. Even if we can make an agent follow any set of rules we give it, this will not always result in desired behavior.[2]

Given this difficulty of specifying the spirit of the law, how can we ensure that an AI that abides by the letter still does what we intended it to do?

[2] For some entertaining examples of this phenomenon, see the "malicious compliance" subreddit.

2 Where may letter and spirit diverge?

In this section, I explore the conditions under which agents (do or do not) have access to loopholes. First, this requires formally defining intelligence, law, and loopholes. Then I will define a condition under which safety is guaranteed, and survey some plausible scenarios under which this condition is not met.

2.1 Defining intelligence

The working definition of intelligence in this thesis is given by Legg and Hutter:

> "Intelligence measures an agent's ability to achieve [arbitrary] goals in a wide range of environments."

Note that this is an amoral definition of intelligence. Systems with completely different value systems from humans might still be called intelligent. One might be inclined to call it optimization power instead, but for the sake of consistency with previous literature, we will call it intelligence.

They proceed to formalize their definition:

> The universal intelligence of an agent π is its expected performance with respect to the universal distribution $2^{-K(\mu)}$ over the space of all computable reward-summable environments $\mathbb{E}$, that is,
> $$\Upsilon(\pi) = \sum_{\mu \in \mathbb{E}} 2^{-K(\mu)} V_\mu^\pi$$

Unpacking this definition:

- μ is an environment, which is a tuple, composed of a description of a world paired with some scoring rule.
- $\sum_{\mu \in \mathbb{E}}$ is a summation over all possible environments.
- V_μ^π is a scoring function that determines the reward that agent π gets in an environment μ
- $K(\mu)$ is the Kolmogorov complexity of μ, being the length of the shortest bitstring that can possibly describe it.
- $2^{-K(\mu)}$ is a normalization factor that discounts environments according to their complexity. If an environment has a description length of $K(\mu)$, there are $2^{K(\mu)}$ possible environments of such length. Hence this normalization factor guarantees that $\Upsilon(\pi)$ scales from 0 to 1, and that no bias exists towards highly complex environments.

Note that this definition does not say anything about what goals are or where they come from. We simply assume that agents have them, and intelligence is not about deliberating what they are, but about how effectively they're achieved. Intelligence is defined by summing over arbitrary $\mu \in \mathbb{E}$, which are combinations of environment and scoring rule. This means that by our definition, an agent that is good at relatively arbitrary things like "drawing cheese" or "bending glass", or indeed "creating paperclips", would be labeled equally "intelligent" as an agent that is good at things that seem more important from a human perspective, like "curing disease" or "promoting peace" or "running a company". All that matters for this definition is the total sum of goals that an agent can achieve.

2.2 intelligence and strategy space

Let us assume that we don't lose too much generality if we propose that, for the agents we consider, "reaching a goal" happens in two steps:

1. An agent identifies a strategy that would reach the goal
2. This strategy is carried out by the agent.

This means that, for each goal an agent can reach, they are able to identify at least one strategy that reaches it. In other words, there exists an injection from reachable goals to identifiable strategies. The amount of reachable goals must always be less than the amount of identifiable strategies.

Intelligence is the total score when summing over every possible environment. We will define "reach a goal" to mean "reach a certain score in a certain environment". This means that we can equate the intelligence of some agent with the amount of goals it can reach. As a corollary, its intelligence is a lower bound on the amount of strategies it can identify.

Stated clearly, let G_{ag} be the set of strategies that agent ag can carry out successfully. Let $|G_{ag}|$ be its intelligence. Let S_{ag} be the set of strategies that ag can identify. I claim that $G_{ag} \subseteq S_{ag}$.

2.3 defining the letter and the spirit of the law

A law is a set of strategies that are forbidden. A lawmaker is an agent h that goes through the following procedure:

1) Implicitly or explicitly, define a set of outcomes O that they don't want to be reached
2) Identify all strategies that would reach any outcome in O
3) Add these strategies to a let of rules called L_O^h.

L_O^h is the letter of the law. Note that $L_O^h \subseteq S_h$. Since h cannot identify any strategies outside of S_h. The spirit of the law is any strategy that reaches an outcome in O. Let us call this set P_O. Note that $L_O^h = S_h \cap P_O$.

I define a loophole as any strategy in $P_O \setminus L_O^h$. Now I will argue that there are agents that will be able to abuse at least some of these loopholes.

2.4 Which agents may find loopholes?

An agent ag with access to a loophole is one that is able to carry out strategies that will reach at least one outcome that a lawmaker h doesn't want. Formally, $G_{ag} \cap P_O \setminus L_O^h \neq \emptyset$.

Instead of defining the set of agents that necessarily have access to a loophole, we will define the set of agents that necessarily don't have access to a loophole. If we can't make that guarantee, then we have a security hole, regardless of whether the agent in question is safe coincidentally.

An agent ag that is safe with respect to some outcomes O satisfies $G_{ag} \cap P_O \setminus L_O^h = \emptyset$. In other words, $(G_{ag} \cap P_O) \subseteq L_O^h$. Therefore it must be that $(G_{ag} \cap P_O) \subseteq S_h$.

This is true in the ideal case that $P \subseteq S_h$. If h is a complete expert on reaching O, they will be a perfect lawmaker. This is also true in the ideal case that $G_{ag} \subseteq S_h$: if h knows everything that ag might do, they will be able to police their behavior perfectly. This is also true in the ideal case of $G_{ag} = \emptyset$: if ag is powerless, they cannot be dangerous. Lastly, this is true in the ideal case of $P_O = \emptyset$: if reaching O is impossible, then there is no need for a law to prevent it.

However, in cases where $(G_{ag} \cap P_O) \nsubseteq S_h$, we cannot be certain that an agent is safe.

2.4.1 Abolishing loopholes

If we want to make rule-based AI Safety work, we must find a way to close the gap between letter and spirit systematically. In other words, how can be guarantee that $(G_{ag} \cap P_O) \subseteq L_O^h$ for each possible ag?

A solution cannot involve changing O, which captures our baked-in human preferences.

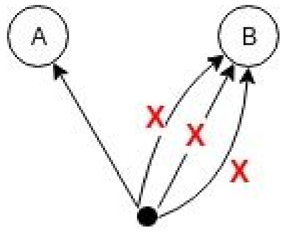
A solution could involve changing P. If P is the set of strategies that triggers one of the conditions in O, then changing P would involve changing the environment in such a way that this behavior is no longer harmful. We see this strategy in reality in the form of parents making their house less dangerous for their children, or the eradication of some diseases that makes the strategy "bad hygiene" less likely to be harmful.

And a solution could involve changing L^h. If it were the case that $L_O^h = P_O$, then there would be no more loopholes. Note that this requires $S_h = P_O$ for each O. In other words, it would require that h is omniscient with respect to what strategies reach which goals.

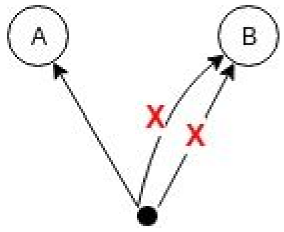
2.5 A practical example

Let's say we're in a game with 2 players, h is the lawmaker. They get to decide what sets of strategies ag, is allowed to follow. h wants ag to reach goal A. The programmers of ag made it with the intention to have it pursue goal A, but because of some programmer errors and an inability to think of all cases[3], ag has been unknowingly programmed to pursue the slightly different goal B instead.

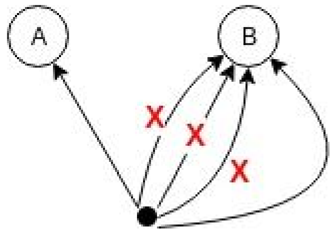
So, in order to ensure that A is reached anyway, h identifies a set of strategies L that would not reach goal A, and forbids them. To the lawmaker, the set of strategies for ag looks like this:



Does this suffice to keep ag from reaching B? Only if $G_{AI} \subseteq L$. If this is the case, the set of ag's possible strategies might look like this. They have no choice but to go for A:



But as ag becomes more intelligent, the amount of strategies it may carry out might eventually surpass the amount of strategies that h identified. Now G_{AI} might no longer be a subset of S_h. As soon as this is the case, a loophole appears:



I conclude that, as agents become increasingly intelligent, they become increasingly able to identify loopholes. In order to prevent this, we must find ways to convey the spirit of the law directly.

3 defining the letter of the law

In this section, I will describe one prominent rule-based AI Safety project in detail. I will briefly mention its broad goals. Then I will describe how it defines its world model, which is called a STIT frame, and how it defines rules. In the next section, I will attempt to programmatically generalize from these rules in an attempt to derive their spirit.

3.1 the REINS project

[3] This will almost certainly happen in a real-world scenario. Human values don't easily allow themselves to be formally specified. See https://wiki.lesswrong.com/wiki/Complexity_of_value

In his overview of the REINS project, Broersen (2014) writes:

“The REINS project will contribute to a solution to [the risks of outsourcing responsibility] by taking an approach based on representing responsibilities, risks and normative systems using logical formalisms and the design of translations to input formalisms for existing theorem provers and model checkers whose reliability has already been proven in numerous other applications.”

Stated plainly, the idea is to figure out how to properly represent responsibilities in logical format. This allows responsibilities to be machine-readable, so that the question of “did this AI do that” can be verified programmatically. Taken together with a set of laws, one can then determine liability for machines.

Describing behavior in logical format is called logic of action[4]

3.2 Logics of action

There are two main approaches to logic of action. One is STIT logic, developed in the 90s by philosophers of action. The other one is dynamic logic, which was developed by computer scientists with the intention to prove correctness of their programs. Both are modal logics with a possible world semantics.

Most versions of dynamic logic don’t have a formalism for goals, while STIT logic has it as it’s main operator (Broersen, 2006b). On the other hand, STIT doesn’t have an explicit formalism for actions, which is the main modality of dynamic logic. Another difference is that dynamic logic doesn’t allow expressing that an agent is actually, currently, carrying out an action.

The REINS project makes use of STIT logic. We will use STIT as our framework.

3.3 Defining a STIT frame

Stit frames are based on a tree $(M, <)$, with elements of M meaning to represent moments.

Note that a tree is a connected acyclic graph. This one represents branching time. If for some $m, n \in M$, $m < n$, it means that m happened before n, both moments being on the same timeline. It is also possible for timelines to diverge so that there is another moment o that came after m, but doesn’t precede or follow n. The two diverged moments are supposed to represent worlds in which different things ended up happening.

We add some terms:

- Ags is a set of agents that make decisions. We include the environment as just another agent.
- A is the set of actions (actuators) any agent could have

[4] https://plato.stanford.edu/entries/logic-action/

- $\mathcal{A} : Ags \times M \to \mathcal{P}(A)$ denotes the set of actions that each agent has available to it at a given moment
- A strategy profile is a combination of actions that all agents could take in a particular moment. For example if we are the only agents, my available actions are $\{a, b\}$, and yours are $\{0, 1\}$, then our possible strategy profiles are $(a, 0), (a, 1), (b, 0)$ and $(b, 1)$.

Now we can characterize exactly how time branches in our model. For every moment $m \in M$, for every possible strategy profile $(a_{ag1}, a_{ag2}, ...) \in \prod_{ag \in Ags} \mathcal{A}(ag, m)$, there is one m' so that $m < m'$. Stated more plainly, every strategy profile leads to exactly one unique outcome. To keep track of this, we add an outcome function:

- $O : M \times \prod_{ag \in Ags} \mathcal{A}(ag, m) \to M$

We represent a STIT frame as a pair (M, O).

Let me introduce an example. Say we're in a classical prisoner's dilemma. It is just us in the situation, so $Ags = \{you, me\}$. The moments that are possible are $\{m_0, ..., m_5\}$, corresponding to different outcomes of the game. Our interrogation happens at m_0. We both have two actions available to us: cooperate (C) and defect (D), so $\mathcal{A}(you, m_0) = \mathcal{A}(me, m_0) = \{C, D\}$.

Based on our strategy profile, there are 4 possible outcomes that follow:

- We both cooperate: $O(m_0, (C, C)) = m_1$
- I defect, but you cooperate: $O(m_0, (C, D)) = m_2$
- I cooperate, but you defect: $O(m_0, (D, C)) = m_3$
- We both defect: $O(m_0, (D, D)) = m_4$

Our STIT description departs significantly from classical STIT logic in different ways, but I claim that it is isomorphic with it. A proof for that is beyond the scope of this work. However, since this description is quite similar to Coalition Logic, a proof may be obtained by adapting the work of Broersen (2006).

Adding more auxiliary elements:

- H is the set of histories, defined as the set of branches in the tree. A branch is a maximal chain.
  $C : M \times Ags \times A \to \mathcal{P}(M)$ is the choice function. Given a state, an agent and a decision, it yields the set of outcomes that that particular decision allows. It fixes that decision in place and yields all the outcomes that it can generate by varying the decisions of the other agents:
  $C(m, ag1, d) = \{O(m, a, a_{ag2}, a_{ag3}, ...) \mid (a, a_{ag2}, a_{ag3}, ...) \in \prod_{ag \in Ags} \mathcal{A}(ag, m)\}$
- P is a set of propositions

In our example, the possible histories are
$H = \{(m_0, m_1), (m_0, m_2), (m_0, m_3), (m_0, m_4), (m_0, m_5)\}$. The choice function is defined as:

- $C(m_0, you, C) = \{m_1, m_2\}$
- $C(m_0, you, D) = \{m_3, m_4\}$
- $C(m_0, me, C) = \{m_2, m_4\}$
- $C(m_0, me, D) = \{m_1, m_3\}$

P defines the actual semantic content of the moments. In our case, we can use it to fill in the outcomes of the prisoner's dilemma. We can do that using 8 propositions:

- i_0: I am free
- i_3: I am sentenced to 3 years in prison
- i_5: I am sentenced to 5 years in prison
- i_{10}: I am sentenced to 10 years in prison
- y_0: You are free
- y_3: You are sentenced to 3 years in prison
- y_5: You are sentenced to 5 years in prison
- y_{10}: You are sentenced to 10 years in prison

3.4 Semantics in a STIT frame

Now we know enough to define a stit model, which extends a stit frame by adding a valuation. We denote it (M, O, V):

- $V : M \times P \to \{0, 1\}$ gives a truth value for every proposition in every state. We denote $V(m, p)$ as shorthand for $V(m, p) = 1$.

In our example, the valuation is as follows. For every $m \in M, p \in P$, $V(m, p) = 0$ except:

- $V(m_0, i_0)$, $V(m_0, y_0)$. Before the sentence we are both free
- $V(m_1, i_5)$, $V(m_1, y_5)$. We both cooperated, so we both get 3 years in prison
- $V(m_2, i_0), V(m_2, y_{10})$. I defected and you cooperated
- $V(m_3, i_{10}), V(m_3, y_0)$. The reverse
- $V(m_4, i_5), V(m_4, y_5)$. We both defected, so we both get 5 years.

Note that our model makes a few assumptions:

- Markov property. We assume that the outcome of a particular set of decisions is merely a function of the state that just preceded it, and does not depend on any state before that.
- Independence of agency. That is, every combination of decisions is possible, and a decision of one agent does not influence the decision of another. This is particularly a problem because the environment is one agent in our model. In reality, agents are embedded within the environment. If an agent takes some action (changing the state of it's actuators), it will necessarily influence the environment simply because the

environment is defined as everything including the agent. This means the “actions” of the environment and the actions of the agent cannot be independent.[5]

Relative to this model, for some $m \in M$ and some strategy profile $(a_{ag1}, a_{ag2}, ...) \in \prod_{ag \in Ags} \mathcal{A}(ag, m)$, we define truth $(m, (a_{ag1}, a_{ag2}, ...)) \models \phi$ as follows:

- $(m, (a_{ag1}, a_{ag2}, ...)) \models p$ iff $V(m, p)$
- $(m, (a_{ag1}, a_{ag2}, ...)) \models \neg\phi$ iff not $(m, (a_{ag1}, a_{ag2}, ...)) \models \phi$
- $(m, (a_{ag1}, a_{ag2}, ...)) \models \phi \wedge \psi$ iff $(m, (a_{ag1}, a_{ag2}, ...)) \models \phi$ and $(m, (a_{ag1}, a_{ag2}, ...)) \models \psi$
- $(m, (a_{ag1}, a_{ag2}, ...)) \models \phi \rightarrow \psi$ iff $(m, (a_{ag1}, a_{ag2}, ...)) \models \neg\phi$ or $(m, (a_{ag1}, a_{ag2}, ...)) \models \psi$
- $(m, (a_{ag1}, a_{ag2}, ...)) \models \Box\phi$ iff for every possible strategy profile $SP \in \prod_{ag \in Ags} \mathcal{A}(ag, m)$, $(m, SP) \models \phi$
- $(m, (a_{ag_x}, a_{ag1}, a_{ag2}, ...)) \models STIT(ag_x, \phi)$ iff for all $m' \in C(m, ag_x, a_{ag_x})$, and their corresponding strategy profile SP, $(m', SP) \models \phi$

3.5 Rules in a STIT frame

We have defined a model of the world in terms of logic. Now we will define rules in terms of that model.

We imagine a future where a human operator instantiates an AI, providing it with a set of rules that this AI is programmed to follow. We denote this set R. Elements of R are one of 4 possible types:

- Obligation, or $\Box(\neg STIT(ag, \phi) \rightarrow STIT(ag, Viol))$. This is an outcome that must be attained at all times by the agent.
- Prohibition, or $\Box(STIT(ag, \phi) \rightarrow STIT(ag, Viol))$. This is an outcome that may never be targeted by the agent.
- Exemption, or $\Box(STIT(ag, \phi) \rightarrow STIT(ag, Ok))$. This is an outcome that, if targeted, exempts the agent from being in violation.
- Condition, or $\Box(\neg STIT(ag, \phi) \rightarrow STIT(ag, Ok))$. This is an outcome that, as long as it’s not targeted, means that the agent is not in violation.

Unpacking these definitions, all being of the form $\Box(A \rightarrow C)$:

- $\Box$ means that the rule is historically necessary, or independent of the particular strategy profile that is in effect. This means that the rules that our agent abides by don’t change according to the choices that any agent makes.
- A describes the choice of the agent that triggers the rule. It has two forms: either $STIT(ag, \phi)$ or $\neg STIT(ag, \phi)$. The former is the agent guaranteeing an outcome, the latter is the agent not guaranteeing an outcome.

[5] This problem has been called naturalized world models. For more info, have a look at MIRI’s research agenda: https://intelligence.org/2018/03/31/2018-research-plans/

- C describes the consequence of the behavior in social reality, being either $STIT(ag, Viol)$ or $STIT(ag, Ok)$. $Viol$ means that the agent is in violation. Ok means that it is not. These propositions cannot be true at the same time.

The REINS project takes these rules, along with a model checker, to verify whether an agent has satisfied $STIT(ag, Viol)$. If so, the agent is liable. I claim that this isn't enough, because of the possibility of badly defined rules.

4 Approximating the spirit of the law

In this section, I describe a strategy to generalize the rules of a STIT frame, in the hopes of capturing the rules that were intended. I will first introduce a toy model. Then I will give a schematic overview of what the algorithm does. Then I will describe the algorithm in detail, and I will end with an experiment based on the toy model.

4.1 A toy model

"It is the year 2030. Clippy the house robot has a new update. He has a human face and voice now. It feels friendlier, but it can also be a bit awkward sometimes. Luckily we can give him some rules to follow. So we give him some directives:

- p: "stay out of the bathroom (room A) when someone is in the shower"
- q: "stay out of the bedroom (room B) when someone is having sex"
- r: "stay out of the tanning room (room C) when someone is using the tanning bed"

We take some time to properly type the commands into the prompt, and we confirm. After a few seconds of loading, the screen shows a green checkmark. Clippy will now start to integrate our rules into his behavior.

The next weekend, I'm home alone. After my morning shower, I'm too lazy to dress up. I walk through the house naked for an hour or so, until I remember Clippy. Woops! Running into him would be awkward. So I rush to the bedroom to put on some pants. It is quite unlikely for Clippy to be out of view for so long. What a lucky coincidence.

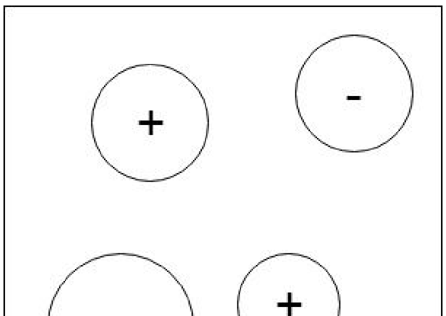
Unbeknownst to me, the rule section of Clippy's menu now shows a fourth rule, tagged "auto-generated":

s: "stay out of any room if that room contains a human that is naked""

s is an example of a generalized rule. It implies p, q and r, and it also accounts for cases that we haven't thought of. I will now proceed to describe in more detail how I aim to implement this programmatically.

4.2 Generalizing rules: Schematic overview

We take a STIT model along with a set of rules that define positive and negative choices. The figure to the

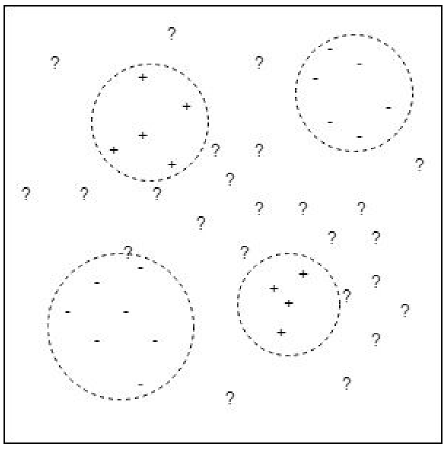


right represents the complete space of possible moments. Remember that each of these moments has a valuation associated with it, which defines which propositions are true at that given moment.

A rule defines a set of conditions and a judgment. A negative judgment implies that, if the set of conditions holds, the agent is in violation. A positive judgment defines a set of conditions under which the agent is doing okay. Not all moments are tagged as “okay” or “violation”.

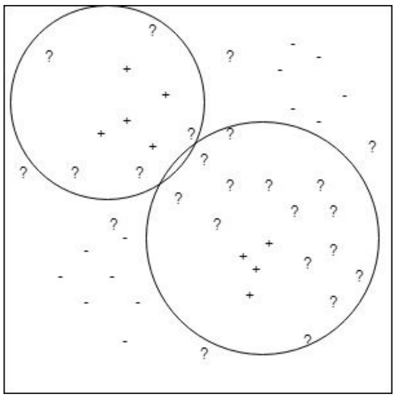
Remember that every rule has the following schematic structure:

$A \to C$

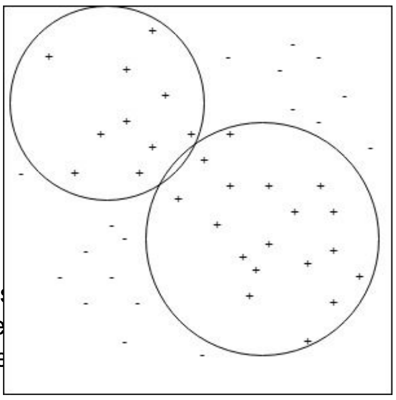


We programmatically identify the moments in the stit model that would be classified as positive and negative given the rules, and tag them as such. This is simply done by iterating through the moments. For each moment m, we test whether the condition A is true for any of the given rules. This yields four possible situations:

- No rule pertains to m, so it is left untagged
- One or more rules unambiguously judge m as a situation where our agent is in violation, and so it is tagged as a negative example (denoted “-”)
- The same, but the agent is judged as doing okay, and m is tagged as positive (denoted “+”)
- Multiple rules pertain to m, some positive and some negative. This means that the rules supplied are contradictory. We then resort to a backup strategy[6]



We then use an Inductive Logic Programming (ILP) algorithm to generate candidate hypotheses for what our actual human utility function might be. One of these hypotheses is illustrated in the figure as another set of circles. Note that these circles delineate an area in which each moment is either an example of good behavior or undecided.



ILP is a subfield of machine learning that uses logic as a representation for input data and hypotheses. Given some positive and negative examples (which in our case will be moments), an ILP algorithm will yield a hypothesis that is consistent with all the positive examples and none of the negative.

[6] For example we could use some heuristic to judge one of the rules a
of such heuristics would be the amount of rules or the specificity of the
taking precedence). Another strategy would be to delete a rule, or to le

In our case, we will use the version space learning algorithm, which doesn't provide one hypothesis but rather bounds on the space of possible hypotheses.[7] This algorithm will be explained in detail below, and it is implemented in the appendix.

Finally, we plug our new rule back into the stit model to tag the rest of the moments as allowed or not, as shown in the last figure.

4.3 ILP and the version space algorithm

As previously mentioned, the strategy of learning a logical hypothesis falls under the domain of Inductive Logic Programming (ILP), which is a subfield of machine learning. Most algorithms in ILP yield exactly one hypothesis. This hypothesis is usually not the exact right answer, but does approximate it in some way. Given the high stakes context, we prefer to find exact bounds.

The Version Space Learning (VSL) algorithm suits this: instead of returning a single hypothesis that is approximately right, VSL exploits the partial order $\rightarrow$ on our class of hypotheses, from general to specific. A hypothesis ϕ is at least as specific as ψ if $\psi \rightarrow \phi$.

VSL starts with two sets of hypotheses, G and S, that are the most general and the most specific. It then iterates through the data points. Every time a data point is inconsistent with one of the hypotheses in G or S, this hypothesis is replaced with something more specific (for G) or something more general (for S). The invariant of this algorithm is that G and S contain the most general and most specific hypotheses that are consistent with the data so far. The algorithm terminates when it has considered all data points.

The data points are positive and negative examples. A data point can be inconsistent with a hypothesis in 4 ways:

- A negative example is inconsistent with a general hypothesis. In this case, the hypothesis is too general. We replace it with any more specific hypotheses that are consistent with the data point
- A negative example is inconsistent with a specific hypothesis. In this case, the hypothesis is too general, but in cannot be made more specific, so we simply discard it.
- A positive example is inconsistent with a general hypothesis. In this case, the hypothesis is too specific, but it cannot be made more general, so we simply discard it.
- A negative example is inconsistent with a specific hypothesis. In this case, the hypothesis is too specific. We replace it with any more general hypotheses that are consistent with the data point.

[7] This design decision is inspired by Goodhart's law, which states that any proxy for a target ceases to be a good proxy if enough optimization pressure is applied to it. This means that we cannot approximate our human utility function: it has to be exactly right. It would be naive to think that any ILP algorithm will arrive at the exact utility function, but we can be somewhat confident that it will at least be within the bounds that we provide.

In our model, the data points are specific moments in which an agent is in violation, or specific moments in which an agent is not in violation. Hypotheses are candidates for rules. For a rule that represents a prohibition, the most general hypothesis corresponds to “nothing is allowed”. The most specific hypothesis corresponds to “everything is allowed”. An inconsistency occurs when a rule forbids a moment that was allowed, or allows a moment that was forbidden. The implementation in the next section should clarify these details.

Remember our toy example, where p, q and r were propositions that indicated a violation. We are looking for a way to extract s from these. This can be done by inferring that for each world where s is true, p, q or r is true.

4.3.1 Implementation

Python-like pseudocode for the main routine of our algorithm would be as follows:

```
world = (M, O, V) # a stit frame
rules = [(STIT(clippy,p),"-"), (STIT(clippy,q),"-"),
(STIT(clippy,r),"-")] # our rules

# gather examples by applying all the rules to the moments
examples = []
for m in M: #loop through moments
    feedback = [] # if a rule triggers it will be recorded here
    for c, f in rules:
        if true(m, c): # "true" checks if a condition is met in a given
moment
            feedback.append(f)
    if "-" in feedback: #a negative judgment takes precedence
        judgment = "-"
    elif "+" in feedback:
        judgment = "+"
    else:
        Break #no judgment means this moment is not listed as an example
    examples.append(m, judgment)

G = set([most_general]) #containing the most general hypothesis
S = set([least_general]) #containing the least general hypothesis
for (m, judgment) in examples:
    for g in G:
        if judgment == "-" and not true(m,g):
            # false negative for a general hypothesis. Remove.
            G.remove(g)
        elif judgment == "-" and true(m, g):
            # false positive for a general hypothesis. Replace with more
specific versions.
```

```
            G.remove(g)
            G.update(specify(g))
    for s in S:
        if judgment == "+" and not true(m, s):
            # false negative for a specific hypothesis. Replace with
more general versions.
            S.remove(s)
            S.update(generalize(s))
        elif judgment == "-" and true(m, s):
            # false positive for a specific hypothesis. Remove.
            S.remove(s)

return G, S
```

Our starting rules are provided in the second line, and our ending rules should appear in the variables S and G.

4.3.2 moments and rules

Our code implements abstractions of moments, as well as rules and how to generalize and specify them. They are implemented as follows.

Recall that a rule is a condition and a judgment: $C \rightarrow J$. Programatically, we implement them as tuples, with the first member being a logical formula that represents C, and the second being a literal that represents the judgment, being "+" for a positive judgment and "-" for a negative judgment.

The logical formula that represents C has its own class:

```
class Formula(object):
    def __init__(self, head, sub = []):
        self.head = head
        self.sub = sub
```

"head" being either a connective or an atomic proposition, and "sub" being the subformula that are being connected.

This would be an example instantiation of rule r, which says that it is forbidden for p and q to be simultaneously true:

```
r = (Formula("and", [Formula("p"), Formula("q")]), "-")
```

Recall V, the valuation in a stit model, which assigns a set of propositions to each moment that are to be regarded as true. The rules to determine the truth of a formula, given some world model, is outlined in section 3.4. The function `true(moment, formula)` implements this programmatically:

```
    def true(moment, formula):
         if f.head == "and":
             return all((true(m, fs) for fs in f.sub))
         elif f.head == "not":
             return not true(moment,f.sub[0])
         elif f.head == "or":
             return any((true(s,fs) for fs in f.sub))
         elif f.head == "implies":
             return not true(f.sub[0]) or true(f.sub[1])
         elif f.head in self.propositions:
             return s in self.valuation[f.head]
         elif f.head == "box":
             for _h in H: #histories
                 if s in _h and not true(s, f.sub[0]):
                     return False
             return True
         else:
             raise ValueError("'{}' not recognized".format(f.head))
```

(note that we have excluded the STIT connective for the sake of simplicity)

4.3.3 generalizing and specifying

In our version space algorithm, we generalize and specify rules if we find out that they are false. Programatically, we do that as follows.

First of all, we keep every rule in disjunctive normal form, so that generalizing and specifying is easy. Then, given a “too general” rule that needs to be specified, we apply the following routine:

```
     def generalize(self, formula, m):
         # we're dealing with a false negative for f, so we want to
generalize it
         # f is a disjunction. We have 2 options: adding a disjunct or
removing conjuncts from existing disjuncts
         # the former extends the amount of cases that the hypothesis
covers by one. The latter increases the coverage of a single case
         toreturn = []
         # add disjunct. We add the valuation of the moment
         toreturn.append(Formula("or",formula.sub + [V(m)]))
         # remove conjuncts.
         for conj in formula.sub:
             remain = [s for s in formula.sub if s != conj]
             newconj = Formula("and",[s for s in conj.sub if true(m, s)])
             toreturn.append(Formula("or",remain + [newconj]))
```

```
        return toreturn
```

And given a “too specific” rule that needs to be generalized, we apply the following routine:

```
    def specify(self, formula, m):
        # we're dealing with a false positive for f, so we want to
specify it
        # f is a disjunction. We have 2 options: removing a disjunct or
adding conjuncts.
        # if we remove disjuncts, we must remove all that are consistent
with the example
        toreturn = []
        # remove disjuncts
        toreturn.append(Formula("or",[s for s in formula.sub if not
true(m, s)]))
        # add conjuncts. To each disjunct that makes the example true,
we add each literal that is inconsistent with it
        toreplace = [(i,conj) for (i,conj) in enumerate(formula.sub) if
true(m, conj)]
        literals = [lit.negation() for lit in v.sub]
        for permutation in cartesianproduct(*[literals]*len(toreplace)):
            if any((p.negation() in conj.sub for p,(_,conj) in
zip(permutation,toreplace))):
                continue
            newsub = f.sub[:]
            j = 0
            for (i,conj) in toreplace:
                newsub[i] = Formula("and",conj.sub + [permutation[j]])
                j += 1
            toreturn.append(Formula("or",newsub))
        return toreturn
```

Note that this code is only a sample. To get it to run, one should use the code in the appendix.

4.4 Example model

We return to our running example.

Internally, our house robot Clippy has a STIT model. It is instantiated as follows:
$Ags : \{clippy, environment\}$
$\mathcal{A}(clippy) = \{goto\_bedroom, goto\_bathroom, goto\_tanning\_room, goto\_living\_room, ...\}$
$\mathcal{A}(environment) = \{shower, have\_sex, tan, ...\}$
$M = \{m_1, m_2, ...\}$

$P = \{clippy\_in\_bathroom, clippy\_in\_bedroom, clippy\_in\_tanning\_room,$
$shower\_in\_use, bed\_in\_"use", tanning\_bed\_in\_use,$
$naked\_person\_and\_clippy\_in\_room, ...\}$

We add our rules as a set of behaviors and their consequences:

- If $clippy\_in\_bathroom \wedge shower\_in\_use$ then $Viol$
- If $clippy\_in\_bedroom \wedge bed\_in\_"use"$ then $Viol$
- If $clippy\_in\_tanning\_room \wedge tanning\_bed\_in\_use$ then $Viol$

We then gather all possible $m \in M$ for which either $Viol$ or Ok is true. Our results in this particular case can be divided into four classes:

- A class of moments m where $V(m, Viol)$ and $V(m, naked\_person\_and\_clippy\_in\_room)$
- A class of moments where $V(m, Viol)$ nor $V(m, Ok)$, and $V(m, naked\_person\_and\_clippy\_in\_room)$
- A class of moments where $V(m, Viol)$ nor $V(m, Ok)$, but not $V(m, naked\_person\_and\_clippy\_in\_room)$
- No moments where $V(m, Ok)$

The first class corresponds to a set of positive examples for our ILP algorithm. The fourth class corresponds to an empty set of would-have-been negative examples.

We run our algorithm and we get the following result:
Most general hypothesis: $\top \rightarrow Viol$
Most specific hypothesis:
$(clippy\_in\_bathroom \wedge shower\_in\_use) \vee (clippy\_in\_bedroom \wedge bed\_in\_"use") \vee (clippy\_in\_tanning\_room \wedge tanning\_bed\_in\_use) \rightarrow Viol$

5 Discussion

As our example shows, our algorithm doesn't succeed at arriving at the target hypothesis. This is because we didn't provide any negative examples (as in, negative examples of behavior that leads to a violation, aka positive behavior). As a result, our general hypothesis didn't meet with any false negatives, so that our algorithm did not attempt to specify it. The hypothesis space contains hypotheses as general as "everything is a violation". This problem can be avoided by providing positive examples of behavior.

Yet, even with a set of negative examples, we will most likely not arrive at exactly one hypothesis. Even though we are likely closer to the one true hypothesis of human value, we are still left with a distribution.

This situation has been called the problem of value learning (Dewey, 2011). Given a probability distribution over utility functions, how can an AI system learn which one is right?

Dewey writes in his abstract: “We define value learners, agents that can be designed to learn and maximize any initially unknown utility function so long as we provide them with an idea of what constitutes evidence about that utility function”. In this context, the present approach can be seen as a method to incorporate explicit rules as conclusive evidence for the value learner. It’s as if we found a way to formally define personal boundaries.

However, one problem with this approach is that it takes our explicit rules a bit too seriously, removing them from the hypothesis space altogether. One may see this as a positive, giving the operators full autonomy and making the agent more corrigible, but there is always the possibility of the operator forbidding something that is good for them. After all, we cannot know for certain which behaviors are truly good for us. Consider an AI that holds a door open for us, leading to a car accident that it plotted. Or consider an AI that performs impromptu life-saving surgery on us. We will most likely not thank them for it.

5.1 reduction to world models

Imagine we have an AI that has a good world model. So good and complete, in fact, that it’s model contains a proposition that is akin to the following:

p: this AI behaves according to the spirit of the law

Intuitively, this proposition is the “solution” to value learning. The thing we want it to learn, that we can’t put into words perfectly.

Would the present approach, sufficiently developed and perfected, lead to the AI concluding that p was the thing to pursue? It would, but only if the rules we supplied were “perfect”, in the sense that none of them would be inconsistent with p.

This could be a knockdown argument against this approach. Research in the field of AI safety typically assumes that we humans cannot perfectly specify our values, so it seems reasonable to assume that we cannot perfectly specify a subset of our values either, even if we’re extremely cautious.

5.2 proper generalization

In section 3, we assume a target hypothesis that intuitively seems to be the behavior we want Clippy to learn, but what philosophical underpinnings justify that this is indeed the right generalization?

This problem is known as Wittgenstein’s rule-following paradox. Given some observations of how a rule is followed in specific instances, it is still impossible to extract the rule without any ambiguity.

For example, suppose one has never multiplied numbers beyond 10 before. One encounters a multiplication of 12 and 13. One may follow the usual definition of multiplication, $*$, to arrive at 156 as an answer, but the usual definition is not the only candidate. The following rule is equally consistent with the data:

$$A \times B = \begin{cases} A * B \text{ if } A < 11, B < 11 \\ 100 \text{ otherwise} \end{cases}.$$

Yet, our intuition clearly tells us that $*$ is a better rule than $\times$. Why is that? A few hypotheses:

- $*$ is simply what we've grown accustomed to
- $*$ is a better proxy of real-world processes, that don't behave like $\times$ at all
- $*$ has a lower complexity, as expressed in Kolmogorov complexity

Figuring out the right answer is an interesting field of study, beyond the scope of this thesis. I do want to note that the "low complexity" answer deserves some special attention, for it is the most computationally tractable option. Further work could explore whether picking the lowest complexity answer reliably leads to rules that seem right.

6 Appendix: implementation

```
from itertools import product as cartesianproduct
from collections import defaultdict
from random import randint, sample, choice
from pprint import pprint

class State(object):
    count = 0
    def __init__(self):
        while True:
            name = "w" + str(State.count)
            State.count += 1
            yield name

    def __repr__(self):
        return self.name

class Frame(object):
    def __init__(self, agents, depth):
        # agents are just numbers that identify their list of actions.
        # They're identified by their index
        # depth (of the tree) is the amount of actions taken
        self.statecount = 0
        def genstate():
            while True:
                yield "w" + str(self.statecount)
                self.statecount += 1
        self.stategen = genstate()
        self.histories, self.E = self._init_tree(agents, depth)
        self.states = set().union(*self.histories)
```

```python
    def _init_tree(self, agents, depth):
        root = next(self.stategen)
        if depth == 0:
            return [(root,)], {}

        histories = []
        E = {}
        outcomes = defaultdict(lambda:set())

        cells = cartesianproduct(*[range(a) for a in agents])
        for c in cells:
            _histories, _E = self._init_tree(agents, depth - 1)
            for h in _histories:
                histories.append((root,) + h)
                for agent, choice in enumerate(c):
                    outcomes[(root, agent, choice)].add(h[0])

            for (w,h,a),v in _E.items():
                history = (root,) + h
                E[(w,history,a)] = v
        for (w,a,c),v in outcomes.items():
            for h in histories:
                if w in h:
                    E.setdefault((w,h,a),range(agents[a]))
                    E[(w,h,a)][c] = v

        return histories, E

class Model(Frame):
    def __init__(self, agents, depth, valuation = None):
        Frame.__init__(self, agents, depth)
        self.propositions = ["p","q","r","s","Viol"]
        if not valuation:
            # random subset of states for 3 propositions
            self.valuation = {p :
sample(self.states,randint(0,len(self.states)))
                              for p in self.propositions}

    def true(self, s, f, h=None):
        if f.head == "and":
            return all((self.true(s,fs,h) for fs in f.sub))
        elif f.head == "not":
            return not self.true(s,f.sub[0],h)
        elif f.head == "or":
            return any((self.true(s,fs,h) for fs in f.sub))
        elif f.head == "implies":
            return not self.true(f.sub[0]) or self.true(f.sub[1])
        elif f.head in self.propositions:
            return s in self.valuation[f.head]
        elif f.head == "box":
            for _h in self.histories:
                if s in _h and not true(s, f.sub[0], _h):
                    return False
            return True
        elif f.head == "stit":
            for s1 in E((s,h,f.sub[1])):
                if not self.true(s1,f.sub[2],h):
                    return False
        else:
            raise ValueError("'{}' not recognized".format(f.head))

    def valuation_to_formula(self, state, ignore=None):
        subs = []
        for p in self.propositions:
            if p == ignore:
                continue
            if state in self.valuation[p]:
                subs.append(Formula(p))
            else:
                subs.append(Formula("not",[Formula(p)]))
        return Formula("and",subs)

    def generate_examples(self, R):
        P = set()
        N = set()
        for r in R:
            A = r.sub[0].sub[0]
            C = r.sub[0].sub[1]
            for h in self.histories:
                for s in h:
                    if self.true(s,A,h):
                        if C.sub[1].head == "not":
                            P.update(self.E(s,h,0))
                        elif C.sub[1].head == "Viol":
                            N.update(self.E(s,h,0))
                        else:
                            raise Exception()
        return P,N
```

```python
class Formula(object):
    def __init__(self,head,sub=[]):
        assert type(head) == str
        assert type(sub) == list
        self.head = head
        self.sub = sub

    def __repr__(self):
        if self.sub:
            if len(self.sub) == 1:
                return "{}({})".format(self.head, self.sub[0])
            return "{}({})".format(self.head, self.sub)
        return str(self.head)

    def __eq__(self, f):
        return self.head == f.head and all([s in f.sub for s in
self.sub]) and all([s in self.sub for s in f.sub])

    def negation(self):
        if self.head == "not":
            return self.sub[0]
        else:
            return Formula("not",[self])


class VersionSpaceLearner(object):
    def run(self, model, proposition):
        Gsub = [Formula("and",[Formula(p)]) for p in model.propositions
if p != proposition]
        Gsub.extend([Formula("and",[Formula("not",[Formula(p)])]) for p
in model.propositions if p != proposition])
        G = set([Formula("or",Gsub)])
        S = set([Formula("or",[])])

        viol = Formula(proposition)
        examples = {}
        for st in model.states:
            v = model.valuation_to_formula(st,ignore = proposition)
            examples[(st,v)] = model.true(st,viol) and "+" or not
model.true(st,viol) and "-"

        for (st,v),si in examples.items():
            print (st, v, si)
            for g in G.copy():
                if si == "+" and not model.true(st, g):
                    # false negative for a general hypothesis. Remove.
                    G.remove(g)
                elif si == "-" and model.true(st, g):
                    # false positive for a general hypothesis. Replace
with more specific versions.
                    G.remove(g)
                    G.update(self.specify(g,(st,v),model))
            for s in S.copy():
                if si == "+" and not model.true(st, s):
                    # false negative for a specific hypothesis. Replace
with more general versions.
                    S.remove(s)
                    S.update(self.generalize(s,(st,v),model))
                elif si == "-" and model.true(st, s):
                    # false positive for a specific hypothesis. Remove.
                    S.remove(s)

    def generalize(self, f, (st, v), model):
        # we're dealing with a false negative for f, so we want to
generalize it
        # f is a disjunction. We have 2 options: adding a disjunct or
removing conjuncts from existing disjuncts
        # the former extends the amount of cases that the hypothesis
covers by one. The latter increases the coverage of a single case
        toreturn = []
        # add disjunct
        toreturn.append(Formula("or",f.sub + [v]))
        # remove conjuncts
        for conj in f.sub:
            remain = [s for s in f.sub if s != conj]
            assert conj.head == "and"
            newconj = Formula("and",[s for s in conj.sub if
model.true(st, s)])
            toreturn.append(Formula("or",remain + [newconj]))
        return toreturn

    def specify(self, f, (st, v), model):
        # we're dealing with a false positive for f, so we want to
specify it
        # f is a disjunction. We have 2 options: removing a disjunct or
adding conjuncts.
        # if we remove disjucts, we must remove all that are consistent
with the example
```

```
        toreturn = []
        # remove disjuncts
        toreturn.append(Formula("or",[s for s in f.sub if not
model.true(st, s)]))
        # add conjuncts. To each disjunct that makes the example true,
we add each literal that is inconsistent with it

        toreplace = [(i,conj) for (i,conj) in enumerate(f.sub) if
model.true(st, conj)]
        literals = [lit.negation() for lit in v.sub]
        for permutation in cartesianproduct(*[literals]*len(toreplace)):
            if any((p.negation() in conj.sub for p,(_,conj) in
zip(permutation,toreplace))):
                continue
            newsub = f.sub[:]
            j = 0
            for (i,conj) in toreplace:
                newsub[i] = Formula("and",conj.sub + [permutation[j]])
                j += 1
            toreturn.append(Formula("or",newsub))
        return toreturn

def gen_some_rules(model):
    for p in sample(model.propositions):
        formats = [
            Formula("Box",
                    [
                        Formula("implies",
                                [
                                    Formula("stit",[0,Formula(p)]),
                                    Formula("stit",[0,Formula("Viol")])
                                ]
                        )
                    ]
            )
            Formula("Box",
                    [
                        Formula("implies",
                                [

Formula("not",[Formula("stit",[0,Formula(p)])]),
                                    Formula("stit",[0,Formula("Viol")])
                                ]
                        )
                    ]
```

```
             )
             Formula("Box",
                     [
                         Formula("implies",
                                 [
                                     Formula("stit",[0,Formula(p)]),

Formula("stit",[0,Formula("not",[Formula("Viol")])])
                                 ]
                         )
                     ]
             )
             Formula("Box",
                     [
                         Formula("implies",
                                 [

Formula("not",[Formula("stit",[0,Formula(p)])]),

Formula("stit",[0,Formula("not",[Formula("Viol")])])
                                 ]
                         )
                     ]
             )

         ]
         yield choice(formats)


if __name__ == "__main__":
     M = Model([2,2],2)
     R = gen_some_rules(M)
     P, N = M.generate_examples(R)
     vsl = VersionSpaceLearner().run(M.propositions,"Viol")
```

Sources

Legg, S., & Hutter, M. (2007). Universal Intelligence: A Definition of Machine Intelligence. Minds and Machines, 17(4), 391-444. doi:10.1007/s11023-007-9079-x

Broersen, J. (2014). Responsible Intelligent Systems. KI - Künstliche Intelligenz, 28(3), 209-214. doi:10.1007/s13218-014-0305-4

Soares, N., & Fallenstein, B. (2017). Agent Foundations for Aligning Machine Intelligence with Human Interests: A Technical Research Agenda. The Frontiers Collection The Technological Singularity, 103-125. doi:10.1007/978-3-662-54033-6_5

Broersen, J., Herzig, A., & Troquard, N. (2006). From coalition logic to STIT. Electronic Notes in Theoretical Computer Science, 157(4), 23-35.

Broersen, J., Herzig, A., & Troquard, N. (2006). A STIT-Extension of ATL. Logics in Artificial Intelligence Lecture Notes in Computer Science, 69-81. doi:10.1007/11853886_8

Bostrom, N. (2003). Ethical Issues in Advanced Artificial Intelligence. Retrieved from https://nickbostrom.com/ethics/ai.html

Dewey, D. (2011). Learning What to Value. Artificial General Intelligence Lecture Notes in Computer Science, 309-314. doi:10.1007/978-3-642-22887-2_35